

JavaTM magazine

By and for the Java community 

LOCAL-VARIABLE TYPE INFERENCE 60 | SCALA 47 | BLOCKCHAIN 36

JANUARY/FEBRUARY 2017

TOOLS

17

POLYGLOT:
MOVING MAVEN
PAST XML

22

INSIDE THE
ARCHITECTURE
OF BUILD TOOLS

29

BUILD YOUR OWN
JVM DEBUGGING
TOOLS



17

POLYGLOT FOR MAVEN: MOVING MAVEN INTO SCRIPTING

A new tool from the developer of Maven enables POM files to be written in Ruby, YAML, Groovy, and other languages.

COVER ART BY I-HUA CHEN

03 From the Editor

The Polyglot Future: As the nature of programming changes, should we add regular coverage of JavaScript to the magazine?

05 Letters to the Editor

Comments, questions, suggestions,
and kudos

08 Events

Upcoming Java conferences and events. And introducing Oracle Code, a new series of 20 free one-day events on polyglot programming, DevOps, and cloud computing.

13 Java Books

Review of *Core Java, Volume II (10th Edition)*, by Cay S. Horstmann

36 [Cryptocurrency](#)

Blockchain: Using Cryptocurrency with Java

By Conor Svensson

Integrating the Ethereum blockchain into Java apps using web3j

22

THE DESIGN AND CONSTRUCTION OF MODERN BUILD TOOLS

By Cédric Beust

A look inside a modern JVM build tool—its architecture and implementation

29

CREATING YOUR OWN DEBUGGING TOOLS

By Andrei Pangin

JDK serviceability technologies allow you into the JVM to solve difficult debugging problems.

47 JVM Languages

Scala: Deeply Functional, Purely Object-Oriented

By Adriaan Moors

A mature, practical, and type-safe language for the JVM

52

Cloud

Java in Containers in the Cloud

By Harshad Oak

Deploy Java apps in Docker containers using Oracle Application Container Cloud Service.

60 Java Proposals of Interest

JEP 286: Local-Variable Type Inference

63

Fix This

By Simon Roberts

Our latest code quiz

21 User Groups

Chicago JUG

68 **Contact Us**

Have a comment? Suggestion?
Want to submit an article proposal?
Here's how.

//from the editor /



The Polyglot Future

As the nature of programming changes, should we cover Java *and* JavaScript?

It's no secret that almost all commercial programs today depend on more than one language. While Java still stretches from the UI all the way to the back-end work (a true end-to-end proposition), its role is frequently complemented by functionality written in other languages. Sometimes, those parts are written using scripts or JVM languages. At other times, a more common scenario unfolds: a Java core surrounded by a UI written in JavaScript. The multilanguage phenomenon—referred to as *polyglot programming*—often implies the use of more than two languages. But if there is just one other language, that language is overwhelmingly JavaScript.

In many ways, JavaScript has been an ever-emerging partner to Java: starting with its name, which Netscape chose to key off of Java's popularity, thereby causing endless confusion for

nontechnical audiences; then the use of both languages in web apps, followed by their joint use in mobile; and of course, then the use of JavaScript as a scripting language for Java, via Nashorn. In addition, tools such as the Google Web Toolkit, which transpiles Java into JavaScript, have furthered the connection between the two languages.

JavaScript is also seeping into back-end computing. This trend is most evident with Node (the former node.js), a JavaScript framework that creates a server-like environment for business logic by providing “an event-driven, non-blocking I/O model that makes it lightweight and efficient.” Its emergence is like Ruby on Rails a decade ago: a convenient solution to small-scale projects. Node is likely to expand its presence in that sector due to its reliance on a widely used language.

PHOTOGRAPH BY BOB ADLER/VERBATIM

ORACLE®



Level Up at Oracle Code

Step up to modern cloud development. At the Oracle Code roadshow, expert developers lead labs and sessions on PaaS, Java, mobile, and more.

Get on the list
for event updates:

go.oracle.com/oraclecoderoadshow

developer.oracle.com

#developersrule



//from the editor /

I should note that the Java community has responded to Node with an excellent framework, Vert.x, that provides all the former framework's capabilities and more and is rapidly gaining adherents. One of its appealing aspects is represented by the .x in its name, which is a direct counterpoint to the .js in Node's original name. While Node is a JavaScript framework, the x in Vert.x signifies support for many languages. Here again we see the advent of polyglot programming.

Oracle employs tens of thousands of developers and interacts with millions more. The company's considered opinion is that the polyglot trend is real, enduring, and important. Underlying this conviction is the belief that the cloud will hasten the adoption of multiple languages due to the ease of setting up and testing out different stacks and different toolchains in the cloud. With this in mind, the company has asked me whether *Java Magazine* should broaden its mission to cover polyglot programming, primarily via the addition of coverage of JavaScript.

It might seem odd to consider this step for a publication with *Java* in the name. But let's put aside the

issue of the name for a moment. Looking at the last two years, we have regularly covered other JVM languages. In this sense, we've already had a regular taste of polyglot coverage in our pages. But the question I've been asked involves a deeper commitment to the polyglot reality, more than just JVM languages: Should we add regular coverage of JavaScript?

As my job is to serve you, I'd like your opinion. Would you like to see the addition of tutorials, perhaps morphing into regular coverage, of JavaScript in the magazine? The guidance I'm specifically looking for is how much JavaScript coverage would help you in your programming work. Feel free to say "absolutely none," or "lots of it," or anything in between. And please include any further thoughts as well as any recommendations you have on the current mix of articles you find in our pages.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
[@platypusguy](https://twitter.com/platypusguy)

A promotional poster for the Oracle Code event. The background is red with a pattern of small white diamonds. A large orange circle in the top right contains the text "ORACLE CODE" in white. Below this, a yellow banner says "Register Now". The main title "Oracle Code" is in large white letters. Below it, a yellow banner says "New One-Day, Free Event | 20 Cities Globally". A dark red box contains the text "Explore the Latest Developer Trends:" followed by a list of topics: DevOps, Containers, Microservices & APIs; MySQL, NoSQL, Oracle & Open Source Databases; Development Tools & Low Code Platforms; Open Source Technologies; and Machine Learning, Chatbots & AI. At the bottom left, it says "Live for the Code" in white. At the bottom right, it says "Find an event near you: developer.oracle.com/code" in white. The Oracle logo is in the bottom right corner.

ORACLE CODE

Register Now

Oracle Code

New One-Day, Free Event | 20 Cities Globally

Explore the Latest Developer Trends:

- DevOps, Containers, Microservices & APIs
- MySQL, NoSQL, Oracle & Open Source Databases
- Development Tools & Low Code Platforms
- Open Source Technologies
- Machine Learning, Chatbots & AI

Live for the Code

Find an event near you:
developer.oracle.com/code

ORACLE





NOVEMBER/DECEMBER 2016

JUnit 5's Status

Thank you for your multiple articles on JUnit in the November/December 2016 issue. The final delivery of version 5 is now expected to be in the third quarter of 2017. We intend to introduce as few breaking changes as possible, especially in the programming model (Jupiter API) that you covered in your articles. We publish [detailed release notes](#) in our User Guide for people upgrading from existing milestones.

We would love to get the remaining work done as quickly as possible. However, we are very short on time at the moment. As you probably know, we had some initial funding through a crowdfunding campaign, but that was merely a start. If there is a company out there with a strong interest in Java and JUnit that would like to help out, we would certainly be open to discussing possible options if it reached out to us.

—Marc Philipp
JUnit 5 team

JUnit 5—What Is It?

I want to know a little more about JUnit 5. What does JUnit 5 actually do?

—Nikhil Deshmukh

Indeed, we did not provide any basic introduction to JUnit, in part because it is an almost universal tool in Java development. To quote the entry in Wikipedia, “JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks.” In a nutshell, it’s the primary tool by which Java developers test their own code before checking it in.

Missing Class

In the article “Implementing Design Patterns with Lambdas” in the November/December 2016 issue, I think you should explain where the class `OnlineBankingLambda` came from, as it is not included in the source code.

—Carlos Carvajal

This article was excerpted from the book Java 8 in Action, whose lead author, Raoul-Gabriel Urma, replies: “For the code example, the learning outcome is really about the method processCustomer itself (the template method pattern) and how it can be parameterized with a behavior. The class OnlineBankingLambda is not important (actually, it doesn’t matter what its implementation is as long as it declares the processCustomer method).

You'll find in the book that its (pre-Java 8) parent class is:

```
abstract class OnlineBanking {
    public void processCustomer(int id){
        Customer c =
            Database.getCustomerWithId(id);
        makeCustomerHappy(c);
    }
    abstract void makeCustomerHappy(Customer c);
}
```

After introducing the refactoring with behavior parameterization, it becomes:

```
abstract class OnlineBanking {
    public void processCustomer(
        int id,
        Consumer<Customer> makeCustomerHappy){
```



```
Customer c = Database.getCustomerWithId(id);
makeCustomerHappy.accept(c);
}
}
```

```
class LambdaOnlineBanking extends OnlineBanking {
    // it's inheriting processCustomer
    // ... more stuff can come here
}
```

Although I'm glad to have access to back issues, I'm puzzled as to how to download them. Could you explain?

—Several readers

Write to us at javamag_us@oracle.com.

06

JRebel

 ZEROTURNAROUND

RELOAD CODE CHANGES
INSTANTLY

TRY IT NOW!

*Get a free
t-shirt! →*





Devoxx US

MARCH 21–23

SAN JOSE, CALIFORNIA

Devoxx US focuses on Java, web, mobile, and JVM languages. The conference includes more than 100 sessions in total, with tracks devoted to server-side Java, architecture and security, cloud and containers, big data, Internet of Things (IoT), and more.

Jfokus

FEBRUARY 6, TUTORIAL

FEBRUARY 7–8, CONFERENCE

STOCKHOLM, SWEDEN

Jfokus is the largest annual Java developer conference in Sweden. Conference topics include Java SE and Java EE, continuous delivery and DevOps, IoT, cloud and big data, trends, and JVM languages. This year, the first day of the event will include a VM Tech Summit, which is an open technical collaboration among language designers, compiler writers, tool builders, runtime engineers, and VM architects. The schedule will be divided equally between traditional presentations of 45 minutes and informal, facilitated deep-dive discussion groups among smaller, self-selected participants. Space is limited, as this summit is organized around a single classroom-style room to support direct communication between participants.

SnowCamp 2017

FEBRUARY 8, WORKSHOP

FEBRUARY 9–10, CONFERENCE

FEBRUARY 11, “UNCONFERENCE”

GRENOBLE, FRANCE

This technical conference held in the French Alps focuses on Java, JavaScript, and software architecture, with sessions presented in French (primarily) and English. A workshop day precedes the two-day conference. An “Unconference” at the end is for hitting the slopes with your fellow attendees.

DevNexus

FEBRUARY 22–24

ATLANTA, GEORGIA

DevNexus is devoted to connecting developers from all over the world, providing affordable education, and promoting open source values. The 2017 conference will take place at the Georgia World Congress Center in downtown Atlanta. Presenters will include Josh Long, author of O’Reilly’s upcoming *Cloud Native Java: Designing*



Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry, and Venkat Subramaniam, author of Pragmatic's Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions.

Voxxed Days Zürich

FEBRUARY 23

ZÜRICH, SWITZERLAND

Sharing the Devovx philosophy that content comes first, Voxxed Days events see both internationally renowned and local speakers converge. Past presentations have included “Bringing the Per-

formance of Structs to Java (Sort Of),” by Simon Ritter, and “Java Security Architecture Demystified,” by Martin Toshev.

Topconf Linz 2017

FEBRUARY 28, WORKSHOPS

MARCH 1–2, CONFERENCE

LINZ, AUSTRIA

Topconf covers Java and JVM, DevOps, reactive architecture, innovative languages, UX/UI, and agile development. Presentations this year include “Java Libraries You Can’t Afford to Miss,” “8 Akka Antipatterns You’d Better Be Aware

Of,” and “Spring Framework 5: Reactive Microservices on JDK 9.”

QCon London 2017

MARCH 6–8, CONFERENCE

MARCH 9–10, WORKSHOPS

LONDON, ENGLAND

For more than a decade, QCon London has empowered software development by facilitating the spread of knowledge and innovation in the developer community. Scheduled tracks this year include “Performance Mythbusting” and “Every Last Nanosecond: Low Latency Java.”

jDays

MARCH 7–8

GOTHENBURG, SWEDEN

jDays brings together software engineers from around the world to share their experiences in different areas such as Java, software engineering, IoT, digital trends, testing, agile methodologies, and security.

ConFoo Montreal 2017

MARCH 8–10

MONTREAL, QUEBEC, CANADA

ConFoo Montreal is a multi-technology conference for web developers that promises 155

presentations by popular international speakers. Past ConFoo topics have included how to write better streams with Java 8 and an introduction to Java 9.

Embedded World

MARCH 14–16

NUREMBERG, GERMANY

The theme for the 15th annual gathering of embedded system developers is Securely Connecting the Embedded World. Topics include IoT, connectivity, software engineering, and safety and security.

JavaLand

MARCH 28–30

BRÜHL, GERMANY

This annual conference features more than 100 lectures on subjects such as core Java and JVM languages, enterprise Java and cloud technologies, IoT, front-end and mobile computing, and much more. Scheduled presentations include “Multiplexing and Server Push: HTTP/2 in Java 9,” “The Dark and Light Side of JavaFX,” “JDK 8 Lambdas: Cool Code that Doesn’t Use Streams,” “Migrating to Java 9 Modules,” and “Java EE 8: Java EE Security API.”

APRIL 2-3, TRAINING
APRIL 3-5, TUTORIALS
AND CONFERENCE
NEW YORK, NEW YORK

This event promises four days of in-depth professional training that covers software architecture fundamentals; real-world case studies; and the latest trends in technologies, frameworks, and techniques. Past presentations have included “Introduction to Reactive Applications, Reactive Streams, and Options for the JVM,” as well as “Microservice Standardization.”

JAX DevOps

APRIL 3 AND 6, WORKSHOPS
APRIL 4 AND 5, CONFERENCE
LONDON, ENGLAND

This event for software experts features in-depth knowledge of the latest technologies and methodologies for lean businesses. The focus is on accelerated delivery cycles, faster changes in functionality, and increased quality in delivery. Conference tracks include agile and company culture, cloud platforms, container technologies, continuous delivery

and automation, microservices, and real-world case studies. The conference is preceded and followed by a day of workshops. There's also a two-in-one conference package that provides free access to a parallel conference, JAX Finance.

Devoxx France

APRIL 5, WORKSHOPS
APRIL 6–7, CONFERENCE
PARIS, FRANCE

Devoxx France presents workshops, tutorials, and keynotes from prestigious speakers, followed by a cycle of eight mini conferences every 50 minutes. You can build your own calendar and follow the sessions as you wish. Founded by developers for developers, Devoxx France covers topics ranging from web security to cloud computing. (No English page available.)

GREAT INDIAN DEVELOPER SUMMIT

APRIL 25–28
BANGALORE, INDIA

The Great Indian Developer Summit (GIDS), now in its 10th year, offers four days of content grouped by theme. April 26

focuses on Java and JVM languages. Other days focus on web, mobile, DevOps, and big data. Register for each day separately.

Riga Dev Days 2017

MAY 15–17
RIGA, LATVIA

The biggest tech conference in the Baltic states, this three-day event is a joint project of Google Developer Group Riga, Java User Group Latvia, and Oracle User Group Latvia. By and for software developers, Riga Dev Days focuses on the most relevant topics and technologies for that audience with more than 50 sessions on Java, web, and cloud programming.

J On The Beach

MAY 17, WORKSHOPS
MAY 18–19, TALKS
MALAGA, SPAIN

JOTB is an international rendezvous for developers interested in big-data technologies. JVM and .NET technologies, embedded and IoT development functional programming, and data visualization will all be discussed. Scheduled speakers include longtime Java Champion Martin Thompson and

Director of Developer Experience
at Red Hat Edson Yanaga.

JEEConf

MAY 26–27
KIEV, UKRAINE

JEEConf is the largest Java conference in Eastern Europe. The annual conference focuses on Java technologies for application development. This year offers five tracks and more than 50 speakers with an emphasis on practical experience and development of real projects. Topics include modern approaches in the development of distributed, highly loaded, scalable enterprise systems with Java, among others.

jPrime

MAY 30–31
SOFIA, BULGARIA

jPrime is a relatively new conference, with two days of talks on Java, JVM languages, mobile and web programming, and best practices. The event is run by the Bulgarian Java User Group and provides opportunities for hacking and networking.



DEVOXXTM UNITED STATES

MARCH 21-23, 2017 | SAN JOSE, CA

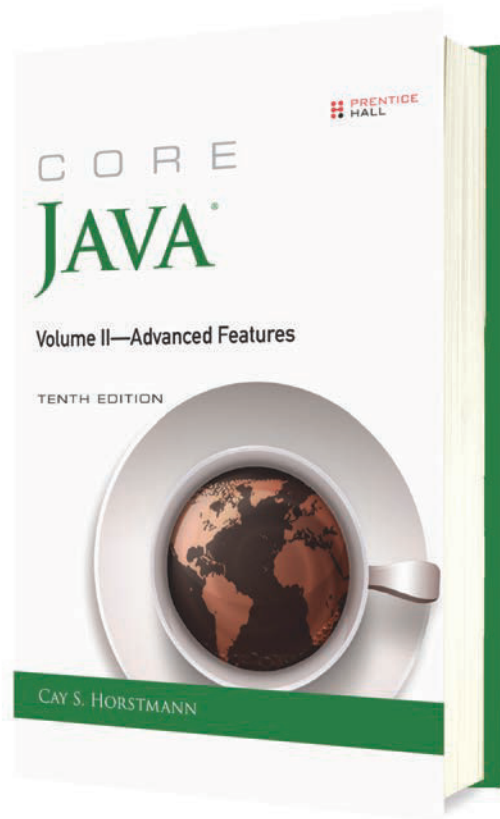
WORLD'S LARGEST
VENDOR INDEPENDENT
DEVELOPER CONFERENCE
IS COMING TO THE USA

WILL YOU BE THERE?



FROM DEVELOPERS, FOR DEVELOPERS

REGISTER NOW | WWW.DEVOXX.US



CORE JAVA, VOLUME II—ADVANCED FEATURES, 10TH EDITION

By Cay S. Horstmann
Prentice Hall

Few programming books have a guaranteed, preallocated place on my bookshelf, but updates to Volumes I and II of Cay S. Horstmann's *Core Java* have occupied the equivalent of front-row center seats on my shelf for years. That is because these two books together represent the clearest, deepest, and most extensive explanation of the Java language and its principal libraries. As with previous editions, each volume tips in at somewhat more than 1,000 pages—which no one could argue is insufficient. However, the sufficiency does not arise from the bulk but the thoughtfulness and depth of the explanations.

A shining example from Volume I, which I previously reviewed, is Horstmann's introduction to lambdas. I've read many explanations of lambdas, but not one explained them so approachably, with easy-to-grasp examples that successively pull

readers along into the complexity of functional interfaces and eventually dump them out at inner classes. In this way, inner classes now make sense in the context of lambdas—rather than explaining inner classes *in order* to demonstrate lambdas.

Along the way, Horstmann throws in tidbits about the historical solutions that preceded lambdas, explains how lambdas differ from similar constructs in other languages, and offers warnings on usage while pointing out where unexpected traps lie. This is how you want to learn a language: deeply and with an author concerned about the reader. This approach also makes *Core Java* an excellent resource for later reference.

While Volume I presents the basics and focuses primarily on the language, the just-released Volume II lays out advanced features of likely interest to professionals. This volume covers the

Java 8 Stream library, advanced I/O, XML, networking, database programming, the completely rewritten Date and Time API, annotation processing, security, native methods, and—curiously—advanced Swing and AWT.

In addition to the admirable text, Horstmann supports his explanations with snippets and, where necessary, complete programs exceeding 100 lines of lucid code.

My only gripe about Volume II is how long it took to come to market. It covers Java SE 8 (not Java 9), which means that more than two years elapsed for it to see the light of day—a long time for a reference and too long for a tutorial. Given that Java 9 is expected to ship this year, you see the problem. But for developers who expect to work with Java 8 for the next few years, there is no better coverage of the language and libraries than in this fine volume. —*Andrew Binstock*



IntelliJ IDEA

Making development*
an enjoyable experience

Get it now

JET
BRAINS



*Java, Groovy, Kotlin, Scala, Android and much more

Java and JVM Tools

POLYGLOT MAVEN [17](#) | INSIDE BUILD TOOLS [22](#) | JVM DEBUGGING [29](#) | LOCAL-VARIABLE TYPE INFERENCE [60](#)

Few things are as exciting to us developers as tools. Nearly everything we do, we do indirectly through the use of tools on which we depend. And in fact, good developers are always characterized by deep knowledge of the tools they use and how best to apply them. With this in mind, in our first article ([page 17](#))

we examine the latest evolution of Maven, called Polyglot Maven, which frees us from creating build files in XML. Polyglot opens the door to using real scripting language for writing build files. This idea is taken one step further in our article ([page 22](#)) on a new build tool, Kobalt, being written in Kotlin, a Java-like JVM language. This article is actually more about the architecture of build tools—what goes on underneath the covers. However, the author’s design uses Kotlin as the language for defining the build. In this way, as he points out, not only do you gain true expressivity in the build file, but your IDE can catch errors as you describe your build. (The increasingly popular build tool

Gradle is moving in the same direction: migrating its build files from Groovy syntax to Kotlin.)

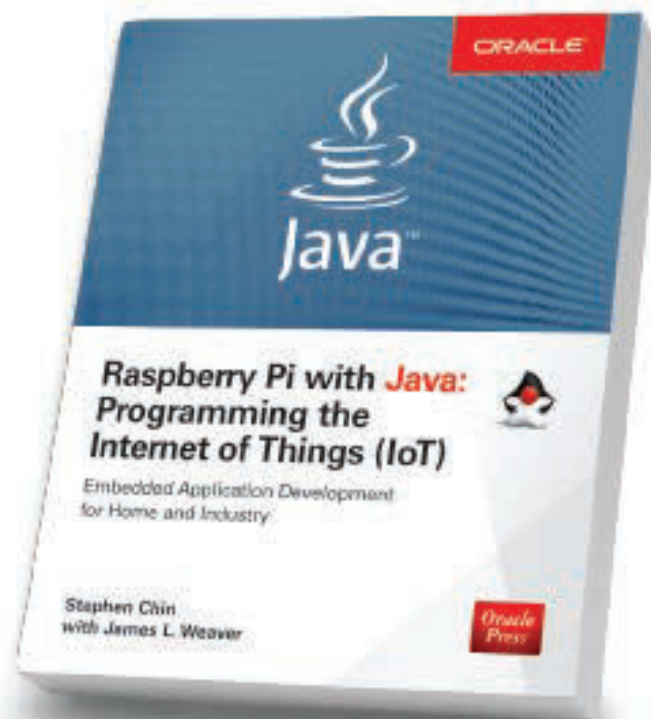
We also cover ([page 29](#)) the uncommon, but not rare, situation in which your debugger does not give you all the information you need to fix your code. We examine a debugging API for the JVM that enables you to extract

information about running processes, view variables and classes, query counters, and get at details you might need. This information is surprisingly easy to access and enables you to write one-off debugging scripts to solve unusual or complex problems.

To complement this information, Oracle’s Brian Goetz discusses lexical language changes supporting local-variable type inference ([page 60](#)). Also, we do a deep dive into Scala ([page 47](#)). Finally, one of the coolest articles ([page 36](#)) we’ve run in a long time: how to get started programming Blockchain, the technology behind cryptocurrencies. Add to this our in-depth language quiz ([page 63](#)), and we hope you’ll find this issue to be packed with useful information.



Written by leading Java experts, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.



Raspberry Pi with Java: Programming the Internet of Things (IoT)

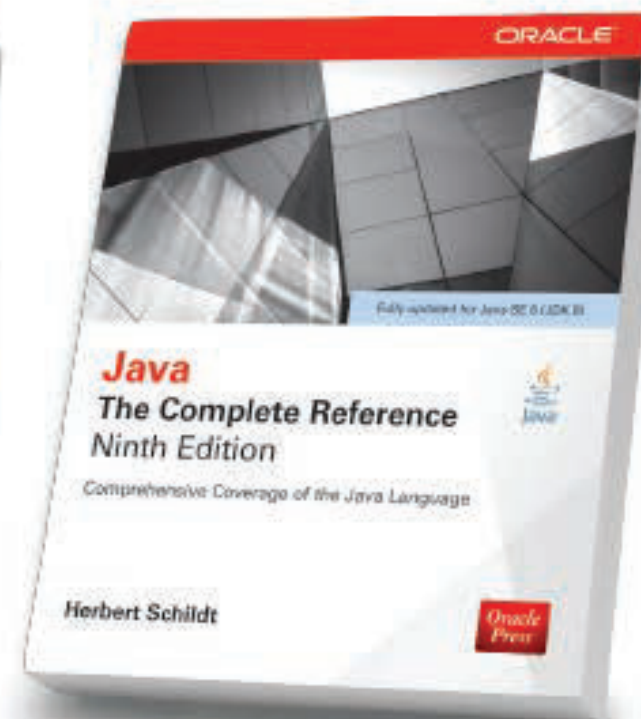
Stephen Chin, James Weaver

Use Raspberry Pi with Java to create innovative devices that power the internet of things.



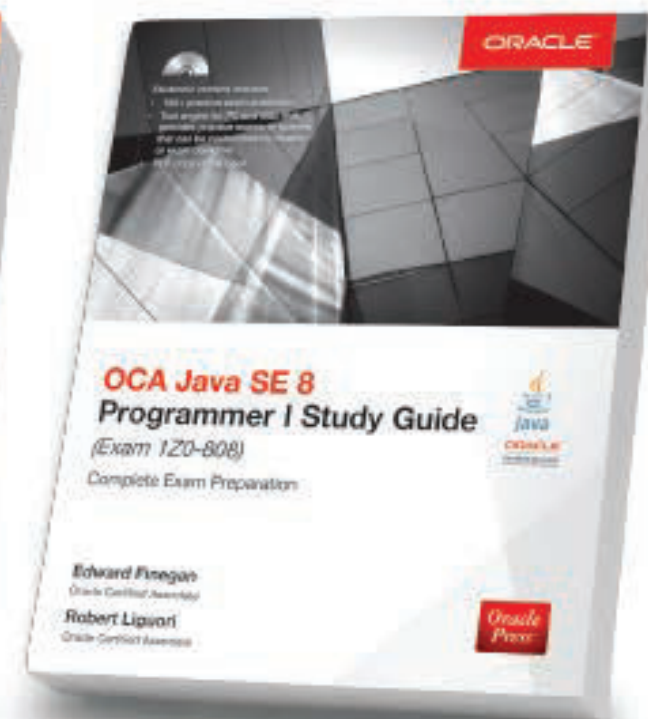
Introducing JavaFX 8 Programming **Herbert Schildt**

Learn how to develop dynamic JavaFX GUI applications quickly and easily.



Java: The Complete Reference, Ninth Edition **Herbert Schildt**

Fully updated for Java SE 8, this definitive guide explains how to develop, compile, debug, and run Java programs.



OCA Java SE 8 Programmer I Study Guide (Exam 1Z0-808) **Edward Finegan, Robert Liguori**

Get complete coverage of all objectives for Exam 1Z0-808. Electronic practice exam questions are included.



JASON VAN ZYL AND
MANFRED MOSER

Polyglot for Maven: Moving Maven into Scripting

A new tool from the developer of Maven enables POM files to be written in Ruby, YAML, Groovy, and other languages.

Apache Maven is the most widely used build tool for the JVM. During the last decade, it introduced new ideas such as dependency management and the usage of repositories to the development ecosystem. The central repository, also known as Maven Central, is the default repository in Maven from which plugins and dependencies are downloaded. It is an exchange hub for open source artifacts in the JVM ecosystem and is used by Maven and other build tools such as Gradle, sbt, Ant/Ivy, and others. As such, Maven continues to be an important tool for the open source community as well as for enterprise users.

XML's Pros and Cons

One common complaint from Maven users over the years is the use of XML as the markup language for the main build file in the project object model (POM or `pom.xml` for short).

XML was a natural choice when Maven was created more than 10 years ago. It has proven to be very useful thanks to its stable structure, well-known syntax, and the powerful tooling available for it. However, times change, and XML is not the latest and greatest option anymore. Today developers demand a terse syntax and features such as inlined scripts. This article shows how [Polyglot for Maven](#), an open source project, provides the ability to use Ruby, YAML, Groovy, and other languages to define the POM.

On the surface, Maven has always used the `pom.xml` file. A minimal `pom.xml` file simply defines an identifier for the project. It comprises the values for `groupId`, `artifactId`, and `version`—the so-called *GAV coordinates* (see **Listing 1**).

■ Listing 1.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example.project</groupId>
  <artifactId>hello-world-demo</artifactId>
  <version>1.0.0</version>
</project>
```

With this simple setup, you can build your JAR archive and as part of that build, you can assemble resources, compile source code and test it, and package the archive. This simple example shows the power of Maven, which derives from several built-in conventions. But it also shows how the verbose nature of the XML file can bury the important information.

In reality, a `pom.xml` file ends up typically being longer due to the inclusion of dependencies, configuration information for plugins, and other aspects about the build and the project at hand.

Picking up the pom.xml file from the project directory, parsing it, and then proceeding to initiate the build by invoking



Performing a build of this project in the yam1 folder of the examples project is identical to performing a build for every other Maven project; that is, just run `mvn clean install`.

■ Listing 3.

```
modelVersion: 4.0.0
groupId: io.takari.polyglot
artifactId: yaml-project
version: 0.0.1-SNAPSHOT
name: 'YAML Maven Love'
properties: {sisuInjectVersion: 0.0.0.M2a}
```

The YAML syntax allows for the declaration of any list with a simple parent node. The example in **Listing 4** shows a list of dependencies using this syntax together with the short syntax for GAV coordinates, including the dependency scope.

■ Listing 4.

```
dependencies:
  - {artifactId: junit, groupId: junit, scope: test,
      version: 4.12}
  - org.codehaus.groovy:groovy-all:2.4.3
```


assumption still needs to be worked out in some dialects. The lessons learned are brought back to the core Maven project, where a separation of this consumption POM from the build POM might provide a path to evolve the native XML POM format in a future version of Maven that won't break backward compatibility for already published artifacts.

A next step the JRuby community has taken is the creation of a [native wrapper for Maven](#). It allows you to use the `rmvn` command, which automatically includes the Ruby extension of Maven, or to run Maven from within a Ruby script (see **Listing 8**).

■ **Listing 8.**

```
require 'ruby-maven'
RubyMaven.exec( '--version' )
```

Conclusion

Interest in and adoption of Polyglot for Maven have been rising since its inception a couple of years ago. It is slowly spreading into the open source community and into enterprises. By providing support for POM files that don't rely only on XML, new ways of expressing builds that are clearer and easier to write are now possible. Take a look at Polyglot for Maven yourself, and please do provide some feedback. </article>

Jason van Zyl is the founder of the Apache Maven project and many other successful open source projects. He started The Central Repository—the largest artifact exchange hub for the open source community—and he leads the development tooling and developer support team at a Fortune 500 company, bringing the results to the open source community at large.

Manfred Moser is an Apache Maven committer and plugin developer. He has trained and helped thousands of Maven users. Moser supports van Zyl as a trainer, author, and developer advocate.

THE CHICAGO JUG



The [Chicago Java Users Group](#) (CJUG) was started by Philip McGlauchlin in 1995 and has grown to nearly 2,000 members in 2016. It is one of the largest JUGs in the United States. The current group of officers includes Java authors, speakers, open source project leads, members of the

JCP, the Apache Software Foundation, and a Java Champion.

CJUG's current focus is on expanding the contributor base to the Java ecosystem in Chicago. To accomplish this, CJUG has been running an Adopt-a-JSR program for the past two years. The program focuses on JSR 366 (Java EE 8) and Java 9. The effort is spearheaded by Josh Juneau and Bob Paulin. This year, the program featured a quarterly call-in discussion.

CJUG also supports membership via an online Slack community called the [Chicago Tech on Slack](#). CJUG has taken a unique approach to sponsorship by constantly moving the meetup to different Chicago companies. This approach means that any developer can experience a broad cross-section of Chicago companies just by attending CJUG's bimonthly meetings.

While CJUG attracts name speakers, its focus is on developing local speaking talent via quarterly lightning talks. The group's current community leader is Java Champion Freddy Guime, who started the [Java OffHeap podcast](#) to provide current news and interviews with Java thought leaders.

CJUG is always looking for new contributors to join and help continue the mission of making Chicago a great place to be a Java developer.



CÉDRIC BEUST

The Design and Construction of Modern Build Tools

A look inside a modern JVM build tool—its architecture and implementation

The JVM has seen many build tools in the past 20 years. For Java alone, there's been Ant, Maven, Gradle, Gant, and others, while in JVM languages, sbt is used for Scala, and Leiningen for Clojure. These tools are run by developers and by back-end systems multiple times a day, yet very little is formally documented about how they are implemented, what functionalities they do and should offer, and why.

In this article, I give an overview of what it takes to create a modern build tool, the kinds of functionality you should expect, and how that functionality is implemented in modern tools. These observations are derived from my experience building software for decades and also from an experimental build tool I am working on called [Kobalt](#), which performs builds for the JVM language Kotlin, [previously described](#) in this magazine.

Motivation

Kobalt was born from my observation that while the Gradle tool was a clear step forward in versatility and expressive build file syntax, it also suffered from several shortcomings, some of them related to its reliance on Groovy. I therefore decided to create a build tool inspired by Gradle but based on Kotlin, JetBrains' language.

Kobalt is not only written in Kotlin, its build files are also valid Kotlin programs with a thin DSL that will look familiar to seasoned Gradle users. (Note that the latest version of

Gradle also adopted Kotlin for its build file syntax, and the Groovy build file is going to be phased out, thus validating the approach taken with Kobalt.)

In this article, I discuss general concepts of build files and demonstrate Kobalt's take on that feature. For readers unfamiliar with Kotlin, you should be able to follow along because the syntax is similar enough to Java's.

Morphology of a Build Tool

The vast variety of build tools available on the JVM and elsewhere share a common architecture and similar basic functionalities:

- The user needs to create one (or more) build files describing the steps required to create an application. The syntax to describe these build files is extremely varied: from simple property files to valid programming language source files and everything in between.
- The build tool parses this build file and turns it into a graph of tasks that need to be executed in a specific order.
- The tool then executes these tasks in the required order. The tasks can either be coded inside the build tool or require the invocation of external processes. The build tool should allow for tasks to do pretty much anything.

One of the oldest build tools on the JVM is Ant, which broke from the previous standard, make, that was in use up until that point. Ant introduced XML as the language for build

a programming language should you ever need them (if, else, classes, inheritance, composition, and so on) is a clear step in the right direction.

Dependencies

The #1 job of a build tool is to execute a sequence of tasks in a certain order and to take various actions if any of these tasks fail. It should come as no surprise that all the build tools I have come across (on the JVM or outside) allow you to define tasks and dependencies between these tasks. However, a lot of tools don't adequately address project dependencies: how do you specify that project C can be built only once projects A and B have been built?

With Gradle, you need to manipulate multiple build .gradle files that, in turn, refer to multiple settings.gradle files. In this design, dependent modules need to be defined across multiple files with different roles (build.gradle and settings.gradle), which can make keeping track of these dependencies challenging.

When I started working on Kobalt, I decided to take a more intuitive approach by making it possible to define multiple projects in one build file, thereby making the dependency tree much more obvious. Here is what this looks like:

```
val lib1 = project {
  name = "lib1"
  version = "0.1"
}
```

```
val lib2 = project {
```

No matter how extensive the build tool is, it will never be able to address all the potential scenarios that developers encounter every day, so it also needs to be expandable.

```
name = "lib2"
version = "0.1"
}

val mainProject = project(lib1, lib2) {
  name = "mainProject"
  version = "0.1"
}
```

The project directive (which is an actual Kotlin function) can take dependent projects as parameters, and Kobalt will build its dependency tree based on that information. All the projects are then sorted topologically, which means the build order can be either “lib1, lib2, mainProject” or “lib2, lib1, mainProject”—both of which are valid.

Also, note that the previous example is a valid and complete build file (and the repetition can be abbreviated further, but I'm keeping things simple for now).

Being able to keep the project dependencies in one centralized place makes it much easier to understand and modify a build, even for a complex project. Using multiple build files in the subprojects' own directories should still be an option, though.

Make Simple Builds Easy and Complex Builds Possible

A direct consequence of the functionality described in the previous section is that the build tool should let you create build files that are as bare bones as possible.

Convention over configuration. Most projects usually contain just one module and are pretty simple in nature, so the build tool should make such build files short, and it should implement as many sensible defaults as possible. Some of these defaults include those shown in Table 1.

With such defaults, the simplest build file for a project should literally be less than five lines long. And of course, the build tool could perform further analysis to do some addi-

tional guessing, such as inferring certain dependencies based on the imports.

Complex builds. Once you get past simple projects, the ability to easily modify a build is critical. Such modifications can be made statically (with simple changes to the build files) or dynamically (passing certain switches or values to the build run in order to alter certain settings). The latter operation is usually performed with profiles—values that trigger different actions in your build without having to modify your build file.

Maven has native support for profiles, but Gradle relies on the extraction of environment values in Groovy to achieve a similar result, which reduces its flexibility. Profiles in Kobalt combine these two approaches with conditionals. You define profiles as regular Kotlin values, as shown here:

```
val experimental = false
val premium = false
```

You can use them in regular conditional statements anywhere in your build file, as shown in the following examples:

NAME	NAME OF THE CURRENT DIRECTORY
VERSION	"0.1"
LANGUAGE(S)	AUTOMATICALLY DETECTED
SOURCE DIRECTORIES	src/main/java, src, src/main/{language}
MAIN RESOURCES	src/main/resources
TEST DIRECTORIES	src/test/java, test, src/test/{language}
TEST RESOURCES	src/test/resources
MAVEN REPOSITORIES	MAVEN CENTRAL, JCENTER
BINARY OUTPUT DIRECTORY	{SOMEROOT}/classes
ARTIFACT OUTPUT DIRECTORY	{SOMEROOT}/libs

Table 1. Sensible defaults for a Java-aware build tool

```
val p = project {
  name = if (experimental) "project-exp"
        else "project"
  version = "1.3"
  ...
}
```

Profiles can then be activated on the command line:

```
./kobaltw -profiles \
    experimental,premium assemble
```

This is an area where having your build file written in a programming language really brings benefits, because you can insert profile-triggered operations anywhere that is legal in that programming language:

```
dependencies {
    if (experimental)
        "com.squareup.okhttp:okhttp:2.5.0"
    else
        "com.squareup.okhttp:okhttp:2.4.0"
}
```

Here, `if (experimental)` refers to the profile specified on the command line.

Performance

You want your build tool to be as fast as possible, which means that the overhead it imposes should be minimal and most of the time building should be expended by the external tools invoked by the build. On top of this obvious requirement, the build tool should also support two important features needed for speed: incremental tasks and parallel builds.

Incremental tasks. For the purposes of build tools, a task is incremental if it can detect all by itself whether it needs to run. This is usually determined by calculating whether

Plugin Architecture

No matter how extensive the build tool is, it will never be able to address all the potential scenarios that developers encounter every day, so it also needs to be expandable. This is traditionally achieved by exposing a plugin architecture that developers can use to extend the tool and have it perform tasks it wasn't originally designed for. A build tool's usefulness is directly correlated to the health and size of its plugin ecosystem.

Interestingly, while OSGi is a respectable and well-specified architecture for plugin APIs, I don't know of any build tool that uses it. Instead, build tools tend to invent their own plugin architecture, which is unfortunate.

This topic would require an entire book chapter of its own, so I'll just mention that there are basically two approaches to plugin architecture. The first one is to give plugins full access to the internal structure of your tool, which is the approach adopted by Gradle (driven and facilitated by Gradle's Groovy foundation).

In contrast, the Eclipse and IntelliJ IDEA development environments and Kobalt expose documented endpoints that plugins can connect to and use in order to observe and modify values in an environment that the build tool completely controls. I prefer this approach because it's statically verifiable and much easier to document and maintain.

Package Management

On top of being a very versatile and innovative build system, Maven introduced what will most likely be a legacy that

These days, nobody has time to go to a website, download a package, and manually install it.

Build tools should be no exception, and they should self-update.

will far outlast it: the repository. I'm pretty sure that even if Maven becomes outdated and no longer used, we'll still be referencing and downloading packages from the various Maven repositories that are available today.

Given the popularity of these repositories, a modern build tool should do the following:

- Transparently support the automatic downloading of dependencies from these repositories (Maven, Gradle, and Kobalt all do this; Ant requires an additional tool).
- Make it as easy as possible to make my projects available on these repositories. Maven and Gradle require plugins and quite a bit of boilerplate in your build file; Kobalt natively supports such uploads.

Auto-Completion in Your IDE

Developers spend several hours every day in their IDE, so it stands to reason that they would expect all the facilities offered by their IDE to be available when they edit a build file. Build tools have been moderately successful at this, frequently offering partial syntactical support.

Interestingly, Maven with its POM file has always been very well supported in IDEs because of its reliance on an XML format that's fully described in an XML schema. The file is verbose, but auto-completion is readily available and reliable, and Maven's schema is a very good example of how to define a proper XML file with very strict rules (for example, no attributes are ever used, so there's never any hesitation about how to enter the data).

The more modern Gradle has been less successful in that area because of its reliance on Groovy and the fact that this language is dynamically typed. Kobalt's reliance on Kotlin for its build file enables auto-completion to work in IntelliJ IDEA, without requiring any special efforts. Obviously, the upcoming Kotlin-based Gradle will enable similar levels of auto-completion as well.



ANDREI PANGIN

Creating Your Own Debugging Tools

JDK serviceability technologies allow you into the JVM to solve difficult debugging problems.

Java is more than just a programming language; it is a comprehensive platform for developing and running application software. One of the most recognized advantages of the Java platform is the large set of serviceability and maintainability features that are especially important for enterprise applications. Besides the numerous built-in tools, there is a lot of third-party software to assist with troubleshooting.

Every decent IDE for Java has a powerful debugger that supports step-by-step execution, breakpoints, watches, and so on. To address performance issues, there are well-known profilers such as Oracle Java Mission Control, JProfiler, and YourKit. Memory leaks are another prevalent problem. There are appropriate tools to deal with memory issues, too—for example, VisualVM and Eclipse Memory Analyzer. General-purpose tools are good for solving typical problems developers often face. However, they do not always fit uncommon situations. I believe most of you know how to use standard tools, so here I discuss how to create new tools customized for your specific cases.

Why Build a Custom Tool?

Imagine a situation in which something has gone wrong with an application on a production server while you are out of the office. It would be helpful to take a heap dump for analysis, but the internet connection might not be good enough to download a multigigabyte dump file. The only thing you can do is run something small remotely. But what tool would you run? Sometimes it is faster to make a specialized tool by

yourself rather than search for an existing one that might or might not help in a particular case.

Another instance where a custom tool can be effective is with the problem of ignored exceptions. It is a common mistake to catch declared exceptions and discard them without handling them. The problem is worse when this happens in a third-party library that you cannot modify. I encountered this bug in a proprietary JDBC driver that did not handle an unexpected error from a database server and, thus, could not properly invalidate a stale connection. Even when you have no control over exceptions in a third-party library, you can still intercept them with a specialized tool. I show how to do that in this article.

If you ever wanted to patch a running application without creating a service interruption, you might be interested in a tool capable of modifying loaded code. Of course, there are commercial solutions that can do the job, but why not fix the problem yourself with just a few lines of code?

The list of tasks that might benefit from a custom tool is endless. Thanks to serviceability components included in Java SE Development Kit (JDK), the creation of such tools is much easier. While each of these components deserves a separate article, the following overview sheds light on what these technologies can do.

jvmstat Performance Counters

Monitoring the JVM is one of the key approaches to ensure that a system works well. Java HotSpot VM provides a huge



amount of telemetry data through `jvmstat` performance counters. This data includes several hundred indicators covering nearly all JVM areas: class loading, garbage collection, multithreading, just-in-time compilation, and more. Despite the name, the tools are not all actually counters, and not all of them are about performance; nevertheless, they are very useful for monitoring JVM health. You might think of performance counters as gauges and dials in the cockpit of an aircraft.

jvmsstat counters are available at no cost; that is, Java HotSpot VM exports them anyway, whether you read them or not. The JVM publishes the telemetry data onto the file system as a memory-mapped file in a temporary directory, often called `/tmp/hspanfdata_{user}/{pid}`, where *{pid}* is a Java process ID. This naming convention makes it possible for tools to find running Java processes in the system.

Fortunately, there is no need to replicate the directory scanning logic, because there is already a convenient Java API for that. Although the `jvmsstat` API is not standard, it is supplied with the standard JDK bundle. You only need to include `{JAVA_HOME}/lib/tools.jar` in the classpath.

Here is how to get the process IDs of all running Java HotSpot VMs in the system:

```
import sun.jvmstat.monitor.MonitoredHost;
...
MonitoredHost host =
    MonitoredHost.getMonitoredHost((String) null);

Set<Integer> processIds = host.activeVms();
```

You can ask the JVM to dump a Java heap, print stack traces, change certain VM flags, load an agent library, and so on.

In this code, `null` stands for the local host. It is also possible to monitor remote virtual machines if a remote host runs the `jstatd` utility. Once you have a process ID, you can obtain an instance of `MonitoredVm` and read its `jvmstat` counters (or monitors). The `Monitor` type can be `Integer`, `Long`, or `String`. For example, a monitor named `sun.rt.javaCommand` contains the main class and the arguments used to start the given Java application. To get all the monitors matching the specified regular expression, use the `findByPattern` method, as shown next:

```
MonitoredVm vm = host.getMonitoredVm(
    new VmIdentifier(processId.toString()));

vm.findByPattern(".*").forEach(monitor -> {
    System.out.println(monitor.getName() + " = " +
        monitor.getValue());
});
```

That simple code lists all available monitors with their values. In its output, you can find interesting metrics that are hardly shown by standard tools. For example:

```
// Total time spent in class initializers
sun.cls.classInitTime = 2545394
// Number of contended synchronizations
sun.rt._sync_ContendedLockAttempts = 55
// Duration of stop-the-world VM pauses
sun.rt.safepointTime = 811588
```

For instance, safepoint time is critical for low-latency applications, because it shows how long the application threads were forcibly stopped by the VM. If you choose to monitor safepoint pauses or any other of the 250+ counters, this is already a good monitoring tool, isn't it? It could be further improved to show a dynamic profile collected over time.

neers for debugging crashes inside the JDK. However, they later realized that it could be helpful for a wider group of developers, and now it is bundled with the regular JDK. Start using the SA by including `{JAVA_HOME}/lib/sa-jdi.jar` in the classpath, but remember that the API is not standard and is subject to change in any future JDK release.

Custom tools typically extend an existing `Tool` class, which is already capable of parsing arguments and attaching to a running VM. You just need to implement custom logic inside the overridden `run` method.

```
import sun.jvm.hotspot.runtime.VM;
import sun.jvm.hotspot.tools.Tool;

public class MyTool extends Tool {

    @Override
    public void run() {
        // Actual implementation
        VM.getVM()...
    }

    public static void main(String[] args) {
        new MyTool().execute(args);
    }
}
```

`VM.getVM()` is the starting point to access Java HotSpot VM internal structures. The next example employs `SystemDictionary` to traverse all loaded classes with their class loaders. A similar technique might be useful in detecting memory leaks related to class loading.

```
VM.getVM().getSystemDictionary()
    .classesDo((klass, loader) -> {
        String className = klass.getName().asString();
```

```
System.out.print(className);

String loaderName = (loader == null)
    ? "Bootstrap ClassLoader"
    : loader.getClass().getName().asString();
System.out.println(" loaded by " + loaderName);
});

That was rather simple. The real power of the SA is to restore
VM structures, either from the memory of a live Java process
or from the core dump of an abnormally terminated pro-
cess when the operating system is configured to create such
dumps. The SA provides the reflection-like API to inspect
Java objects and to extract the required fields. Unlike the
reflection, which works from within the same process, the
SA reads memory of a different process or parses a core dump
file. Tools based on this feature can do impressive tricks such
as stealing private keys from a running web server. The fol-
lowing code scans the heap of a target process looking for
the instances of java.security.PrivateKey and printing
their contents.

Klass keyClass = VM.getVM().getSystemDictionary()
    .find("java/security/PrivateKey", null, null);

VM.getVM().getObjectHeap()
    .iterateObjectsOfKlass(new DefaultHeapVisitor() {
@Override
public boolean doObj(Oop obj) {
    InstanceKlass c = (InstanceKlass) obj.getClass();
    OopField f = (OopField) c.findField("key", "[B");
    TypeArray key = (TypeArray) f.getValue(obj);
    key.printOn(System.out);
    return false;
}
}, keyClass);
```

```
Klass keyClass = VM.getVM().getSystemDictionary()
    .find("java/security/PrivateKey", null, null);

VM.getVM().getObjectHeap()
    .iterateObjectsOfKlass(new DefaultHeapVisitor() {
@Override
public boolean doObj(Oop obj) {
    InstanceKlass c = (InstanceKlass) obj.getClass();
    OopField f = (OopField) c.findField("key", "[B");
    TypeArray key = (TypeArray) f.getValue(obj);
    key.printOn(System.out);
    return false;
}
}, keyClass);
```

The SA needs no cooperation from the target JVM, and there is no way to protect against SA interactions. This is not a reason to worry, though. The SA typically requires root privileges to attach to a running process. Also, keep in mind that the target JVM remains suspended while the SA is attached.

So far, I have focused on JDK internal APIs. If you are looking for a more standard way to build your own tool, consider using the JVM tool interface.

JVM Tool Interface

The JVM tool interface (JVM TI) is a standard programming interface designed especially for debugging, monitoring, and profiling software intended to run on top of the JVM. The best thing about JVM TI is its public specification, which is not tied to any particular implementation. It is not required that every JVM provide all JVM TI functionality; however, most popular JVMs do.

The interface is exposed through the header file `jvmti.h`. JVM TI-based tools, called agents, are typically written in C or C++. They run within the same process and communicate with the JVM directly by calling JVM TI functions. The interface looks somewhat similar to Java Native Interface (JNI), so if you have ever written JNI code, you will easily grasp the principles of using JVM TI.

An agent may start at JVM bootstrap (when specified in `-agentlib` or `-agentpath` JVM arguments), or it can be loaded later at runtime using the Dynamic Attach mechanism. To support these options, an agent should define one or several entry points:

The JVM tool interface (JVM TI) is a standard programming interface designed especially for debugging, monitoring, and profiling software intended to run on top of the JVM.

- `Agent_OnLoad`, which is called automatically by the JVM early at startup time
- `Agent_OnAttach`, which is called whenever the library is loaded at runtime

The first thing an agent typically does is get the reference to the JVM TI environment (`jvmtiEnv`), which is necessary for calling JVM TI functions.

```
#include <jvmti.h>
```

```
JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM *vm, char *args, void *unused) {
    jvmtiEnv *jvmti;
    vm->GetEnv((void**)&jvmti, JVMTI_VERSION_1_0);
```

```
// Initialization code
```

```
return 0;
```

}

JVM TI has functions for everything debuggers usually do. You can manage threads, walk through their stacks, iterate through the Java heap, query local variables, set breakpoints, manipulate Java classes, intercept native methods, and do many other things. Besides that, an agent may subscribe to event notifications: the JVM will invoke a provided callback function whenever an event occurs.

The access to JVM TI functionality is capability-based; that is, an agent must explicitly request the capabilities it is going to use. Most of the capabilities are available at runtime, but some can be requested only during the `OnLoad` phase (the time when no classes have been loaded and no bytecodes have been executed). For example, the `can_access_local_variables` capability is available only at startup, because the JVM needs to disable certain optimizations beforehand in order to retain information about all local variables.

The following example requests a capability to generate exception events and sets up the callback to receive notifications about all thrown Java exceptions, both caught and uncaught.

```
jvmtiCapabilities capabilities = {0};
capabilities.can_generate_exception_events = 1;
jvmti->AddCapabilities(&capabilities);

jvmtiEventCallbacks cb = {0};
cb.Exception = ExceptionCallback;

jvmti->SetEventCallbacks(&cb, sizeof(cb));
jvmti->SetEventNotificationMode(
    JVMTI_ENABLE, JVMTI_EVENT_EXCEPTION, NULL);
```

The callback function receives all details about an exception: a thread, a method, and the bytecode index for the thrown exception. The callback also has a reference to the JNI environment. This means you can invoke any JNI function from within. For instance, you can use JNI to call `Throwable.printStackTrace()`. Thus, an agent will print all the exceptions, including ignored exceptions, just before they are caught.

```
void JNICALL ExceptionCallback(
    jvmtiEnv *jvmti, JNIEnv *env, jthread thread,
    jmethodID method, jlocation location,
    jobject exception, jmethodID catch_method,
    jlocation catch_location)
{
    jclass cls = env->FindClass("java/lang/Throwable");
    jmethodID print_method = env->
        GetMethodID(cls, "printStackTrace", "()V");
    env->CallVoidMethod(exception, print_method);
}
```

You can do a lot more useful things with the JVM TI. Besides exceptions, it is possible to trace class loading, garbage collection, lock contention, thread activity, and more.

JVM TI is often confused with the Java debugger agent. There is a popular misconception that JVM TI compromises security and degrades the performance of Java applications. However, the Java Debug Wire Protocol (JDWP) agent is just one example of a JVM TI-based tool; the technology itself does not imply security or performance consequences. Whether an application will suffer from agent overhead solely depends on what the agent does and which capabilities it requests. Consider JVM TI as a sort of extension to JNI. This technology is definitely worth trying.

Changes in JDK 9

All the technologies discussed previously, including private APIs, will remain functional in the upcoming JDK 9. However, the new module system imposes certain restrictions on how you can access these APIs. No longer will `tools.jar` and `sa-jdi.jar` be separate libraries. JDK 9 serviceability features are supplied in the dedicated modules. **Table 1** shows the location of key JAR files in Java 9.

By default, applications cannot access an API from the modules that do not export packages externally. In order to use the private APIs, you need to explicitly break the encaps-

FEATURE	MODULE	PUBLIC
JVMSTAT PERFORMANCE COUNTERS	jdk.jvmstat	NO
DYNAMIC ATTACH API	jdk.attach	YES*
INSTRUMENTATION AP	java.instrument	YES
SERVICEABILITY AGENT	jdk.hotspot.agent	NO

* com.sun.tools.attach.VirtualMachine is accessible from outside, but sun.tools.attach.HotSpotVirtualMachine is not.

Table 1. The location of serviceability APIs in Java 9

Blockchain: Using Cryptocurrency with Java

Hardly a day goes by without a mention of blockchain in the technology or financial press. But what's all the fuss about this technology, and how can you work with it from your Java applications? Before I talk about a library, [web3j](#), that makes interaction possible, let me explain what blockchain is and how it works.

Blockchain technology started with the cryptocurrency Bitcoin. Bitcoin emerged in 2008, and although it was not the first proposed cryptocurrency, it was the first that was completely decentralized, requiring neither a central authority for issuance nor transaction verification. Bitcoin maintains a distributed ledger containing details of all Bitcoin transactions. All Bitcoin ownership is derived from these ledger entries, known as the Bitcoin blockchain. Transactions on the blockchain are generated using a simple scripting language.

Since then, several other blockchain technologies from different groups have emerged; however, Ethereum is pres-

The Ethereum blockchain is driven by the established cryptocurrency Ether. Ether is the second-largest cryptocurrency after Bitcoin with a market capitalization of approximately US\$1 billion dollars, versus Bitcoin's US\$10 billion capitalization.

What Is a Blockchain?

A blockchain is a *distributed ledger technology* (DLT), a term that has emerged for describing technologies such as blockchain that provide decentralized storage of data. Not

Gradle:

Maven:

```
compile ('org.web3j:core:1.0.9')
```

```
<dependency>
  <groupId>org.web3j</groupId>
  <artifactId>core</artifactId>
  <version>1.0.9</version>
</dependency>
```

Place the following code in a runnable class, which displays the Ethereum client version:

```
// defaults to http://localhost:8545/  
Web3j web3 = Web3j.build(new HttpService());
```

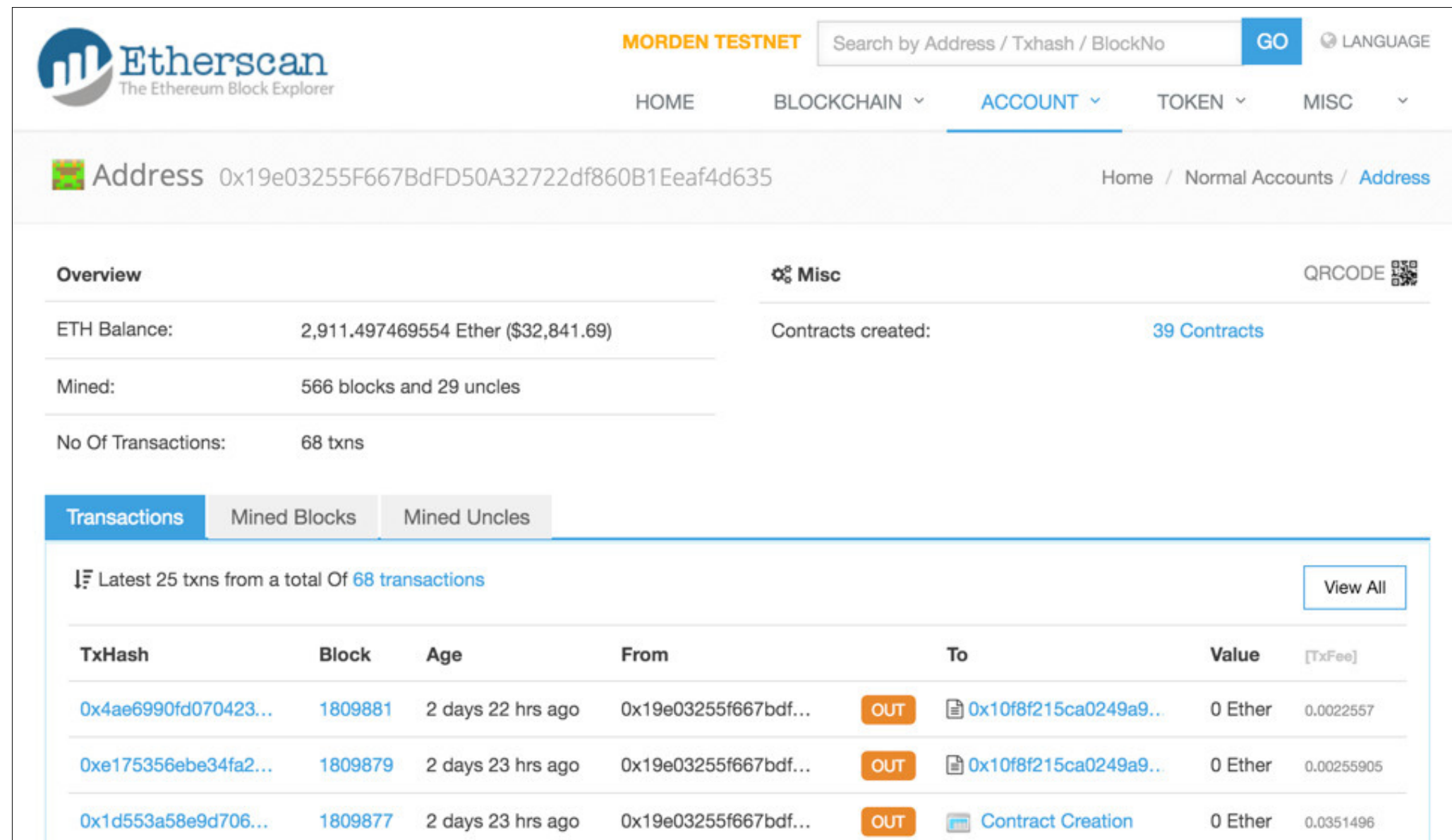


Figure 2. Ethereum wallet address balance on Etherscan

1. The transfer Ether request is submitted to web3j.
2. A transaction message is submitted to an Ethereum client.
3. The client verifies the transaction, and then:
 - a. Propagates the transaction on to other Ethereum nodes
 - b. Takes a hash of the submitted transaction and sends this to the client in a synchronous HTTP response

The diagram illustrates the Ethereum transaction flow, divided into three main sections: **web3j**, **Ethereum Core**, and **Ethereum Network/Blockchain**.

web3j API: The user initiates a transaction (Transfer Ether, Create Contract, or Call Contract) through the API. The API sends the transaction details (To address, Value, Nonce, Gas price, Gas limit) to the Core layer. The API also receives a Transaction Receipt from the Core layer.

Ethereum Core: The Core layer contains the **Transaction** object and the **Signed Transaction** object. The Transaction is signed using the **Wallet Private Key** to create the Signed Transaction. The Signed Transaction is then sent to the Network layer. The Core layer also includes a **Transaction Manager** that polls the Blockchain for the **Transaction status** and returns a **Transaction Receipt** to the API.

Ethereum Network/Blockchain: The Network layer consists of **Ethereum Clients** and **Miners**. The Signed Transaction is sent to the Network layer, where it is propagated to all clients. Clients can **Accept/Reject** the transaction. If rejected, it is sent back to the Core layer. The transaction is then mined into a **New Block** in the Blockchain layer. The Blockchain layer shows a sequence of transactions (Transaction 1, Transaction 2 (our transaction), Transaction 3, ..., Transaction n) within a block. The **Mines** process is indicated by an arrow pointing to the Blockchain layer.

f

public methods available on the smart contract.

web3j requires these two files in order to generate the smart contract wrappers for working with the smart contract in Java.

I generate the greeter wrappers using the web3j command-line tools, this time with the solidity command (full paths have been shortened for brevity):

```
$ ./web3j-1.0.9/bin/web3j solidity generate \
build/greeter.bin build/greeter.abi \
-p org.web3j.javamag.generated -o src/main/java/
```

This command will create the class file `org.web3j.example.generated.Greeter`, which wraps all the smart contracts' methods so they can be called from Java:

```
public final class Greeter extends Contract {
    private static final String BINARY =
        "6060604052604051610269380380610269833981...";

    private Greeter(
        String contractAddress, Web3j web3j,
        Credentials credentials,
        BigInteger gasPrice, BigInteger gasLimit) {
        super(
            contractAddress, web3j, credentials,
            gasPrice, gasLimit);
    }
}
```

...

```
public Future<Utf8String> greet() {
    Function function = new Function("greet",
        Arrays.<Type>asList(),
        Arrays.<TypeReference<?>>asList(
            new TypeReference<Utf8String>() {}));
```

```

        return executeCallSingleValueReturnAsync(
            function);
    }

    public static Future<Greeter> deploy(
        Web3j web3j, Credentials credentials,
        BigInteger gasPrice, BigInteger gasLimit,
        BigInteger initialValue,
        Utf8String _greeting) {
        String encodedConstructor =
            FunctionEncoder.encodeConstructor(
                Arrays.<Type>asList(_greeting));
        return deployAsync(
            Greeter.class, web3j, credentials,
            gasPrice, gasLimit,
            BINARY, encodedConstructor, initialValue);
    }
}

```

```
public static Greeter load(
    String contractAddress, Web3j web3j,
    Credentials credentials,
    BigInteger gasPrice, BigInteger gasLimit) {
    return new Greeter(
        contractAddress, web3j, credentials,
        gasPrice, gasLimit);
}
```

I can now both deploy and call the smart contract:

```
Credentials credentials =
    WalletUtils.loadCredentials(
        "my password", "/path/to/walletfile");
```

```
Greeter contract = Greeter.deploy(
    web3, credentials, BigInteger.ZERO,
```



```
// print the total supply issued
Uint256 totalSupply = contract.totalSupply().get();
System.out.println("Token supply issued: " +
```

```
totalSupply.getValue());

// check your token balance
Uint256 balance = contract.balanceOf(
    new Address(credentials.getAddress()))
    .get();
System.out.println("Your current balance is: w3j$" +
    balance.getValue());

// transfer tokens to another address
TransactionReceipt transferReceipt =
    contract.transfer(
        new Address("0x<destination address"),
        new Uint256(BigInteger.valueOf(100))).get();
```



ADRIAAN MOORS

Scala: Deeply Functional, Purely Object-Oriented

A mature, practical, and type-safe language for the JVM

Scala is a pragmatic, object-oriented JVM language that integrates a comprehensive set of functional programming (FP) features, such as pattern matching and immutable collections, using a compact syntax with powerful type inference. It has gained widespread adoption in many different industries, with some codebases in the millions of lines. Thanks to Apache Spark, Scala is also widely used in the data science community.

Interoperability with Java is a core feature of Scala, and with version 2.12, the language now includes the features of the Java 8 platform. For example, Scala 2.12 compiles lambdas in the same way as Java 8. Similarly, a Scala trait compiles to a Java interface with default methods.

Scala's Origins

Scala was originally developed by Martin Odersky and his research group at the École Polytechnique Fédérale de Lausanne (EPFL), a leading Swiss technical university. Before designing Scala, Odersky codesigned Generic Java (GJ), which brought generics from FP to Java, and he developed a compiler for it. Sun Microsystems adopted the GJ compiler as the standard javac compiler from version 1.3 on (with its support for generics enabled in Java 5).

Meanwhile, Odersky and his research group started work on Scala, releasing version 2.0 in 2006. Around 2007, prominent internet startups such as Twitter and Foursquare started

to use Scala in production, and adoption has grown ever since. In 2011, the company Typesafe was founded to drive commercial adoption of Scala as the language for multicore and cloud computing, using the Akka middleware framework.

Today, Typesafe is known as Lightbend. My team at Lightbend continues to develop the Scala compiler and library in collaboration with an active open source community. A third of the commits in the latest release came from the community. Other vendors are investing in Scala as well. For example, JetBrains' Scala plugin for IntelliJ IDEA has been downloaded almost 5 million times.

The Scala Philosophy

In this article, I explain some of Scala's core concepts and illustrate them with small code snippets. I hope to convince you that, while some developers consider Scala a big language, it actually has a small purely object-oriented core so flexible that it can take many different shapes.

The key idea behind Scala's design is that functional and object-oriented programming are complementary and, thus, combine well into a single language. While FP is often associated with complicated types and deep math, the true appeal of FP in Scala is that it streamlines implementing common tasks in a way that is easier to understand and to scale up. The popularity of lambdas in Java 8 is a good example of this.

Scala aims to be regular and concise, with few restrictions or exceptions. In a definition, all redundant parts are optional.

tion or deep mathematical reasoning. This is not how functional programming is interpreted in Scala. Eventually, code must have an effect on the outside world, whether that code is functional or object-oriented. However, unneeded or unexpected side effects usually adversely affect maintainability and scalability—in *both* functional and object-oriented programming. Scala gives you a choice between “pure” programming, which favors immutable variables and recursion, and more-imperative programming, which uses mutable variables and `while` loops.

Ultimately, Scala is a purely object-oriented language. This means that, conceptually, every value in Scala is an object, and every operation is a method call. Other high-level “conveniences” of the language, such as functions, are translated into this core by expanding them to method calls behind the scenes.

This capability makes it easy to write domain-specific languages (DSLs) in Scala. The use of such DSLs has been implemented successfully by Apache Spark (for data science), Slick (for database queries), and the sbt build tool (for declarative build definitions).

One aspect of Scala's deep integration of FP is simply to translate a function application $f(x)$ to the method call `f.apply(x)`. A function is modeled as an object with an `apply` method, and any object that has this method is, thus, eligible for this convenience.

Following a unification proposed in the Eiffel programming language as the uniform access principle, users of the improved class will not notice the change from `def` message to `val` message. You could even transparently opt for a lazy `val`, which is computed on its first access and then cached. This also generalizes to mutable fields, which are modeled as a getter and setter.

The following snippet declares an abstract, mutable variable in a trait, deferring its implementation in terms of its getter and setter to a subclass (perhaps adding validation). I've omitted the implementation using the `???` method, which is defined in the standard library to throw a `NotImplementedError`.

```
trait T {  
  var v: Int  
}  
  
class Sub extends T {  
  def v: Int = ???  
  def v_=(x: Int): Unit = ???  
}
```

A trait may also define a concrete `val` or `var` member, which the compiler implements automatically in the trait's subclasses.

In keeping with the mantra that every method invocation targets an object, Scala avoids the `static` keyword, and instead offers direct support for the singleton design pattern

When processing data, it's often convenient to bundle a few values together in a tuple.

Because tuples are so common, Scala ships with a set of TupleN case classes with N fields.

through object definitions. It's common practice to define a class and an object of the same name at the same time, and what would be static methods in Java become methods on the class' so-called "companion object."

Here, I define a singleton instance of the C class, instead of creating a new instance explicitly:

```
scala> object o extends C("Hi!")
defined object o

scala> println(o.message)
Hi!
```

Because it doesn't introduce new members, the object definition corresponds closely to lazy `val o = new C("Hi!")`.

Perhaps you've spotted another aspect of Scala's uniformity: definitions are always introduced by their keyword (one of `val`, `var`, `def`, `object`, `trait`, `class`, or `type`), possibly prefixed by some modifiers (such as `lazy` or `private`). Then come the name and the signature, and finally is the right-hand side.

Case Classes and Pattern Matching

As another example of how Scala nudges you toward immutable design, it's very convenient to immediately store constructor arguments in an immutable `val` (or, if you prefer, in a `var`). To make this even more convenient, both `val` and `new` keywords can be omitted when defining a case class:

```
scala> case class C(x: Int)
defined class C
```

Case classes model immutable structured data. They are called *case classes* because they are so easy to use with the `case` keyword when pattern matching, as in this example:

Java in Containers in the Cloud

There has been tremendous interest in application containers during the past couple of years. Application containerization comes with the promise of “build once, run anywhere” in an isolated environment that consumes significantly fewer resources than a virtual machine.

your application code and the server pretty much provides and figures out most other things, using a mix of conventions and configuration in your application.

Containerized applications by their very nature seem suited to public cloud deployments. However, even for traditional Java EE deployments, there are many platform-as-a-service (PaaS) cloud solutions that do a great job of running your Java EE in the cloud. In most cases, you don't need to worry about the underlying hardware, resources, or servers. I discussed these deployments in previous articles in *Java Magazine*, most recently "[Getting Onboard Oracle Java Cloud Service](#)," where I showed how to use Oracle Java Cloud Service for Java EE applications.

There's also a lot of expertise and tooling in place for the established Java EE server-based approach. The container market is still in a state of flux.

Once you're sure that containers are the way to go for your application, you can decide on the kind of container solution to adopt. Let's take a closer look at Oracle's solution.

Before diving into Oracle Application Container Cloud Service, let's quickly go over some things to consider if you're looking to switch to containers from established Java EE deployment servers.

First, it's important to consider the ease of use of Java EE today and the smart, self-contained, and out-of-the-box nature of modern Java EE tools and servers.

Containerized applications are lightweight because they ship with only what they need and nothing else. This means that you need to know exactly which resources, dependencies, and versions to include, which can take some getting used to, especially for Java EE developers accustomed to using application servers that provide everything an application might require. With most current Java EE applications, you provide

About Oracle Application Container Cloud Service

Oracle Application Container Cloud Service is a simple solution that runs your application in its own Docker container in the Oracle Cloud. It currently supports Java and JVM languages, PHP, and Node.js. It promises fast, easy-to-use, self-service provisioning of your applications in isolated container environments in the Oracle Cloud.

The service provides integration with other cloud services offered by the company, including Oracle Database

Cloud Service and Oracle Java Cloud Service. It also offers continuous integration and development if integrated with Oracle Developer Cloud Service.

Although applications in Oracle Application Container Cloud Service run in a Docker container based on Oracle Linux, the service is intended for you to run your applications—not for you to run your Docker containers.

A key benefit of Oracle Application Container Cloud Service over running Docker containers yourself is that for your Java applications, the service comes set up with Oracle Java SE Advanced, which includes Flight Recorder diagnostics and profiling capabilities. Oracle will take care of Java patching and upgrades. The service also handles load balancing and scaling your application. By using it, you get to operate at a higher level of abstraction and don't have to worry about the underlying Java setup, the Oracle Linux base, or the Docker container.

How to Deploy to Oracle Application Container Cloud Service

To use Oracle Application Container Cloud Service, you don't need to code your application any differently. Understanding the service is mostly about packaging, deploying, and managing your application.

Instead of spending time on elaborate code snippets in this article, I'll pick up one of the sample applications in the service documentation and see how you can go about tweaking it a bit and then packaging and running it with Oracle Application Container Cloud Service. I use NetBeans to make the application easier to understand.

Let's begin by [downloading the code](#) for the “Java SE 8: Creating a Web App with Bootstrap and Tomcat Embedded for Oracle Application Container Cloud Service” tutorial. It's a simple standalone web application that uses servlets, JavaServer Pages (JSPs), the Bootstrap front-end framework, and an embedded version of Tomcat.

If you're a Java EE developer used to deploying WAR and EAR files on application servers, you might wonder why we need an embedded Tomcat server. It's worth pointing out here that the Oracle Application Container Cloud Service environment has Java running on it, but there is no server there to run your Java EE application. It's up to you to package the server as part of your application.

Unzip the downloaded code and you will find a Maven project and the requisite project files. I use NetBeans to better understand the project and its constituent parts, especially how the project is configured for deployment on Oracle Application Container Cloud Service. Most IDEs support Maven projects, so you could use some other IDE. If you prefer, you could even use Maven commands from the command prompt.

Using NetBeans and Maven

Open the NetBeans IDE and select File -> Open Project from the menu. Next, select the directory where you extracted the code. NetBeans will figure out that it's a Maven project and handle it accordingly.

As shown in **Figure 1**, you have web pages, a few Java classes, dependencies, and the Maven pom.xml file. The project object model contains project information and the configuration that is used by Maven to build the project. Open the

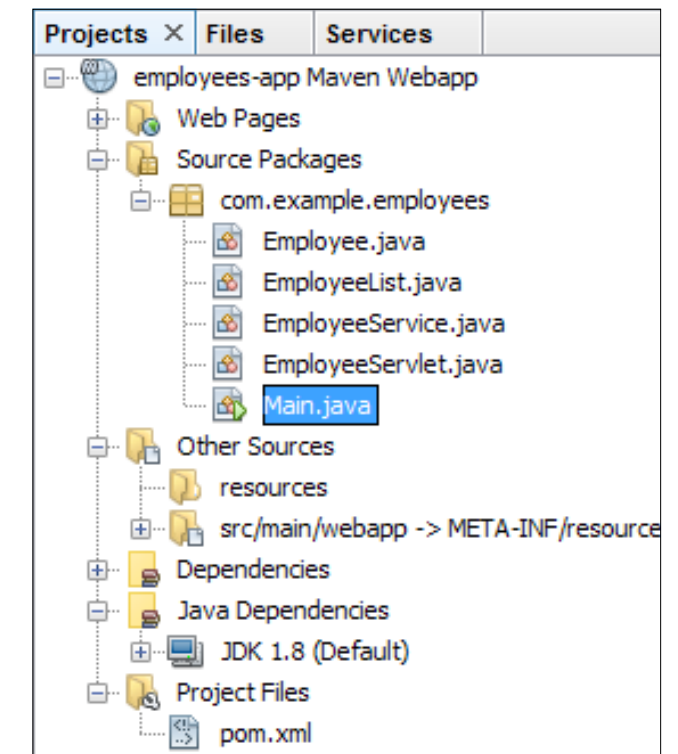


Figure 1. NetBeans Project view

pom.xml file in NetBeans, which offers a Graph view of the file that can help better show the project dependencies.

Now open the Source view and you will see the XML. Let's start from the top and look at some of the key tags for our project.

As shown in **Listing 1**, the Tomcat version is specified as 7.0.57. Tomcat provides different versions based on which specifications are supported. While the creators of the sample application have used version 7.0.57, you can use the latest release of version 7. However, running the application with Tomcat 8.x or 9.x will require additional tweaks.

■ Listing 1.

```
<properties>
    <tomcat.version>7.0.57</tomcat.version>
</properties>
```

The dependencies are shown in **Listing 2**. Except for JSP Standard Tag Library, they're derived by Maven from the dependency on the Tomcat version specified in **Listing 1**.

■ Listing 2.

```
<dependencies>
  <dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-core</artifactId>
    <version>${tomcat.version}</version>
  </dependency>
  . . .
  <dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jsp-api</artifactId>
    <version>${tomcat.version}</version>
  </dependency>
  <dependency>
    <groupId>jstl</groupId>
```

```

        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
</dependencies>

```

Further down the file, as shown in **Listing 3**, are the tags for the Maven Assembly Plugin.

■ Listing 3.

```
<configuration>
  <descriptorRefs>
    <descriptorRef>
      jar-with-dependencies
    </descriptorRef>
  </descriptorRefs>
  <finalName>
    ${project.build.finalName}-${project.version}
  </finalName>
  <archive>
    <manifest>
      <mainClass>
        com.example.employees.Main
      </mainClass>
    </manifest>
  </archive>
</configuration>
```

The `mainClass` here is specified as `com.example.employees.Main`. This declaration is important because this is the class that will be executed when the packaged JAR file is run using the `java -jar` command. So let's look at this `mainClass` next, in Listing 4.

■ **Listing 4.**

```
public class Main {
    public static final Optional<String> PORT =
```

```
Optional.ofNullable(System.getenv("PORT"));
public static final Optional<String> HOSTNAME =
    Optional.ofNullable(System.getenv("HOSTNAME"));

public static void main(String[] args)
    throws Exception {
    String contextPath = "/" ;
    String appBase = ".";
    Tomcat tomcat = new Tomcat();
    tomcat.setPort(Integer.valueOf(
        PORT.orElse("8080") ));
    tomcat.setHostname(
        HOSTNAME.orElse("localhost"));
    tomcat.getHost().setAppBase(appBase);
    tomcat.addWebapp(contextPath, appBase);
    tomcat.start();
}
```

```
        tomcat.getServer().await();
    }
}
```

This code picks up the port and host name from the environment and starts the Tomcat server.

The rest of the code in the project is all about building the actual web application and not directly important for executing in the Oracle Application Container Cloud.

Build the Application

Before you can deploy to Oracle Application Container Cloud Service, the application must be built. Right-click the project name and click Build. You will see in the log that Maven gets the requisite JAR files, copies, compiles, and packages the application such that the executable JAR file is created in the target directory. This file is what is often referred to as a *fat JAR* or an *uber JAR*, because this JAR file includes your code as well as the dependent libraries. The generated JAR includes a manifest.mf file, which specifies `com.example.employees.Main` as the main class to run the JAR file.

If you run this JAR file on your local machine using the `java -jar employees-app-1.0-SNAPSHOT-jar-with-dependencies.jar` command, you will start a Tomcat server on your local machine with your application running on it. Go to `http://localhost:8080` in your browser, and you will get a page as shown in **Figure 2**.

Because our intent is to run the application on Oracle Application Container Cloud Service, there is still some work to be done.

manifest.json Configuration

To upload and deploy the application to the cloud service, you must include a `manifest.json` file in the application archive. The archive could be a `.zip`, `.tgz`, `.tar`, or `.gz` file. In the `manifest.json` file, you need to state the Java version to use and the file

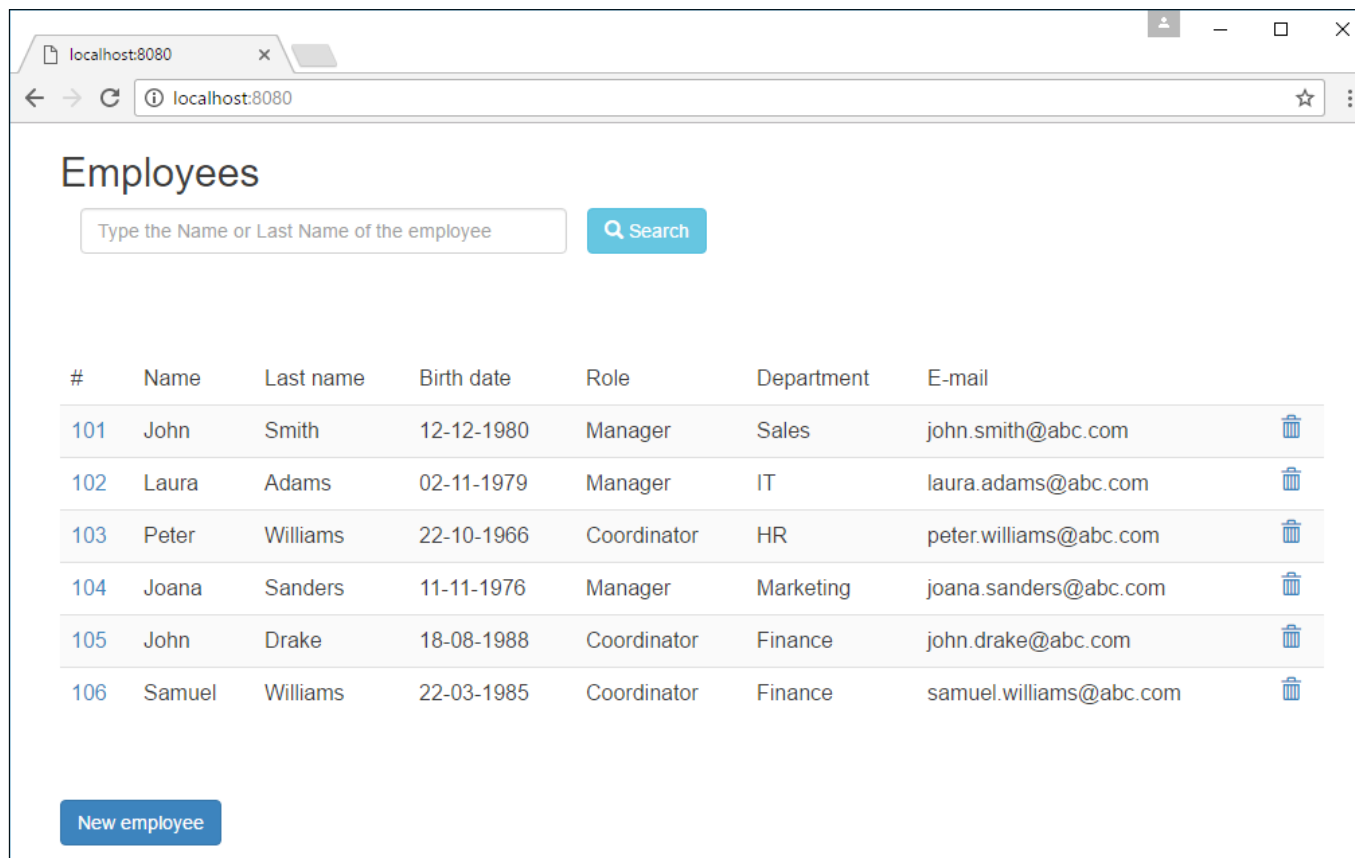


Figure 2. Running the web application




Modify Resources

Figure 4 displays the number of instances and the allocated memory. One of the key features of containerization is that you can easily add identical new containers as required. Change the number of instances to any number between 1 and 16, and within minutes you get identical containers up and running. The existing containers will continue running

uninterrupted while this process is under way.

The memory can be set from 1 GB to 20 GB. Note that modifying the allocated memory will lead to all containers being restarted. Because scaling up the memory requires a restart of all containers and potential downtime, Oracle Application Container Cloud Service provides a Rolling Restart option that stops and restarts container instances


Applications / EmployeeApp

URL: <https://EmployeeApp-harshad.apaas.us2.oraclecloud.com>

Overview

3
Instances

Deployments

0
Service Bindings

0
User-defined Variables

Administration

0
Updates Available

1
Logs

0
Recordings


Updates
Logs
Recordings

As of Nov 25, 2016 5:58:07 PM UTC

You can download logs for your application instances from your Oracle Storage Cloud Service account.

Instance: web.1

1 Logs

Name	Size	Last Uploaded
 server.out.zip	674 B	Nov 25, 2016 5:37:55 PM UTC

Get Log

Instance: web.2

No Logs

Name	Size	Last Uploaded
------	------	---------------

Get Log

Instance: web.3

No Logs

Name	Size	Last Uploaded
------	------	---------------

Get Log

▶ Logs Capture History

Figure 5. The Administration options, showing the Logs tab

In a short period of time and without much effort, the web application is running in multiple, identical, isolated containers in the Oracle Cloud.

In this article, we have used an existing application to try out Oracle Application Container Cloud Service. However, you could create a new Maven project or update an existing project to run your application using Oracle Application Container Cloud Service. The key things you need to do to run your web application using Oracle Application Container Cloud Service

Harshad Oak is a Java Champion and an Oracle Ace Director. He is the founder of IndicThreads and Rightrix Solutions. He is the author of *Pro Jakarta Commons* (Apress, 2004) and has written several books on Java EE. Oak has spoken at conferences in many countries.

Oracle Developer Cloud Service

Local-Variable Type Inference: var and maybe val

More than 2,500 people participated in two rounds of surveys on [JEP 286](#). Overall, the responses were strongly in favor of adding local-variable type inference; 74 percent were strongly in favor, with another 12 percent mildly in favor. 10 percent thought it was a bad idea.

The written comments had more of a negative bias, but this shouldn't be surprising; people generally have more to say in disagreement than in agreement. The positive comments were very positive; the negative comments were very negative. While there were some passionate arguments against, the numbers speak loudly: this is a feature that most developers want. (It is the most frequently repeated request of developers coming to Java from other languages.) So no matter what happens here, some people are going to be very disappointed.

When given a choice, the most popular syntax choice was “val and var,” with “var only” as the second choice. But when asked how they felt about these choices, there was a divergence: more people liked “val and var,” but more people

hated it, too. Although people expressed strong preferences for their favorite syntax, it's important to remember that syntax is just the surface. Language features like this have a lot of complexity under the hood, even when they look simple.

The biggest category of negative comments regarded worries about readability (although most of these came from folks who have never used the feature; those who have used it in other languages were overwhelmingly positive). It is a well-established core design principle of the Java language that reading code is more important than writing code; but plenty of folks assumed that this feature would inevitably lead to less-readable code. Of course, it's easy to construct straw-man examples, such as

```
var x = y.getFoo()
```

to support the belief that this feature would harm readability. But if you dig deeper, you realize that the readability problem here stems from the fact that `x` is just a poorly chosen variable name. (Having a manifest type might make up for the programmer's laziness, but it would be better to just choose a good variable name in the first place.)

Like any convention, the use of `var` can do a lot of damage if not applied properly. But the Java team believes that, when it is used properly, readability can actually be *enhanced*. The reason for this is that it moves the variable name into a more predictable place in the code. Consider a block of locals:

```
UserModelHandle userDB = broker.findUserDB();
List<User> users = db getUsers();
Map<User, Address> addressesByUser =
    db.getAddresses();
```

In most cases, the variable names are more important than anything else on these lines, because they capture the role of the variable in the current program. And in these examples, the variable names are not so easy to visually pick out from the code above—they're stuck in the middle of each line, and they occur at a different place on each line.

Using inferred types brings the variable name prominently into the reader’s view. If the block above were rewritten with inferred types, as follows,

```
var userDB = broker.findUserDB();
var users = db.getUsers();
var addressesByUser = db.getAddresses();
```

then the true intent of the code pops out much more readily, because the variable names are almost right up front, in a predictable place. The lack of manifest types is not an impediment, because good variable names were chosen.

Mutability

Many comments were not so much about the use of type inference, but about mutability. A lot of people like the idea of reducing the ceremony associated with finality. We like this idea, too.

An initial exploration of this feature used inference only for effectively final locals. But after working with a prototype, it was immediately clear how this violated the “bring the variable names front and center” imperative described earlier. Mutable locals would stick out badly:

```
var immutableLocal = ...
```

```
var anotherImmutableLocal = ...
var alsoImmutable = ...
LocalDateTime latestDateSeenSoFar = ...
var thisOneDoesntChangeEither = ...
```

This irregularity was visually jarring to readers of the code. Many people would prefer that we use `val` to mean `final var`, as Scala does, and assumed that they were so similar that readers could mostly ignore the difference if they wanted to. But usability experiments suggested that some people found the subtle difference between `var` and `val` in large blocks of locals, as in the example above, to be distracting. (Others found the more different `var` and `let`, as Swift uses, also distracting.)

The conclusion drawn by the Java team is that while immutability is important, in reality, local variables are the least important place where developers need more help making things immutable. The biggest risk of mutability is data races, but local variables are immune to data races. And the majority of local variables are effectively final anyway. Where more help is needed in encouraging immutability is for *fields*,

but applying type inference there would be foolish.

Further, the `var/val` distinction offers more leverage in other languages than it would in Java. In Scala, for example, all variables—locals and fields alike—are declared with `val name : type`, where you can omit the `: type` if you want inference. So, not only is mutability orthogonal to inference, but the power of the `var/val` distinction is greater because it is used in multiple contexts. However, Java would use it only

The desire to reduce the ceremony of immutability is certainly well taken. However, in this case, it is pushing on the wrong end of the lever.

Quiz Yourself

To start the new year off right, I've put together some more problems that simulate questions from the [1Z0-809 Programmer II exam](#), which is the certification test for developers who have been certified at a basic level of Java 8 programming knowledge and now are looking to demonstrate more-advanced expertise.

Question 1. Given this code:

Which of the following are true? Choose two.

- e.** The code would compile if the modifier `static` were removed from line n1.

Question 2. Given this code:

And this code:

What is the result? Choose one.

- a. Mélina
Sonia
Jaqueline
Alaïs

- At line n4: `) {`
- At line n5: `}`
- c. Insert the following:
 - At line n3: `try (`
 - At line n4: `) {`
 - At line n5: `}`
- d. Insert at line n5: `rs.close(); statement.close();`



These rules mean that the code shown in the question can't work, because it has an inner class attempting to declare a static variable that doesn't qualify as a constant variable. As a result, option A must be incorrect. Also, nothing in the

- Insert the following:
At line n1: `try (`
At line n3: `) {`
At line n5: `}`
- Insert the following:
At line n2: `try (`

documentation suggests that this rule varies based on accessibility, so options B and D are incorrect, too.

That leaves only options C and E as the two correct answers. I should explain why these two are correct, not simply accept the elimination of all the others (even if doing that would have been sufficient for Sherlock Holmes).

Considering option C, if you make the declaration final, you actually have achieved a constant variable. Recall that the definition of a constant variable required a primitive or String, that the variable be final, and that it be initialized with a constant expression. I should justify the assertion that you have a constant expression. Section 15.28 of the specification, “Constant Expressions,” describes the criteria for this, and one of the first items in a fairly long list is the String literal. So, option C is correct.

Option E actually describes the case where the field is reverted to an instance field rather than a static one. There are no restrictions on a regular instance field in an inner class, so option E is also correct.

There's just one thing left to consider, and that is why this rule exists. It probably seems a little arbitrary. I don't have insight into the minds of those who put this together, but I do have an explanation that makes reasonable sense and might serve to make the rule seem sensible—and, therefore, easier to remember—even if it's not actually the logic that led to the rule.

With an inner class—that is, a nonstatic inner class—it’s as if the class itself is a member of the enclosing instance. That would require that every new outer instance create a new inner class, just like a new instance creates a new copy of every field. In this case, the inner class is based on the same source code as all the other inner classes. Each inner class would have its own version of any static members, but how would you refer to those static members? You’d need to distinguish which class you’re referring to when you want to use one of these fields. That could be complex, ugly, error-prone,

and—as history has shown—unnecessary. (Java has survived with this restriction since the inception of inner classes in Java 1.1.)

I'll reiterate that this perspective might not be what the designers had in mind, but they did call these things inner *classes* and not inner *objects*. Either way, I hope the mental model serves to help you remember that statics are not permitted in inner classes. Nested classes, however, have no such restriction.

Question 2. The correct answer is option C. This question investigates the behavior of sets in the Java APIs. The most fundamental behavior of a set is that it does not permit duplicate entries. In the API documentation for the `Set` interface, this requirement is expressed in terms of the `equals` method: specifically, that no two non-null entries may return true when one is tested against the other using the `equals` method, and also that at most one null item is permitted. To do this, before actually adding a new item, the set must test to see if the offered item is already in the set. A simple check of every item in turn would be very time-consuming, particularly as the number of items in the set grows. Therefore, the concrete implementations provided by Java seek to improve the speed of this check.

In `TreeSet`, the items are added to the set in order, and this means that the set can use a binary search to locate an item (or determine that it's not present). A binary search is much more efficient, particularly as the number of items in the set grows. However, in the example in the question, the ordering is determined solely by the `int` field called `order`, and this means that the two objects with `order` equal to 9 are at the same point in the ordering, even though they have clearly

Caution should be exercised when using a comparator capable of imposing an ordering inconsistent with equals.

different values. The Java documentation refers to this as “ordering that is *inconsistent with equals*.” The documentation for `TreeSet` warns: “Note that the ordering maintained by a set. . . must be *consistent with equals* if it is to correctly implement the `Set` interface.”

The effect is that `TreeSet` treats those two objects that have the value 9 in the `order` field as being equal (in fact, the implementation of `TreeSet` never uses the `equals` method when considering the items in the set). As a result, the second item with `order` equal to 9 is considered a duplicate. In this particular example, there is no override of the `equals` method, which strictly means that every object is considered unique, and even objects created with identical field values should properly be permitted to be simultaneous set members.

Because of this, the example considers that the two items with the names Méлина and Jaquelina are duplicates. As a result, they cannot both be in the set at the same time. Therefore, options A and B must be false.

The next consideration is exactly how a `Set` behaves when presented with a duplicate. Does it reject the new item or replace the existing item? This information allows you to decide if option C or option D correctly defines the contents of the set. This consideration is resolved by the documentation for the `add` method of the `Set` interface, which states: “Adds the specified element to this set if it is not already present If this set already contains the element, the call leaves the set unchanged.”

Because of this, you know that the first of the two items that have the same value for **order** will remain in the set, and the second will be rejected. Therefore, Mélina remains, and Jaqueline does not make it into the set. This means that option C is possible, but option D must be wrong.

In Java 7, the JDBC API was extended to make good use of the try-with-resources feature.

Finally, option E suggests that the order might not be predictable. This is an interesting issue. On one hand, a `Set` differs from a `List` in that it does not honor the order in which items are added. But, the reason that no such guarantee is offered is that the `Set` implementation is at liberty to use an internal order that speeds up the search for duplicates. In the case of `TreeSet`, the order will be the order specified for use in the tree-building mechanism. In this example, that's simply the order defined by the `compareTo` method of the `Ordered` class. At this point, you could reasonably object to a question that appears to depend on an implementation detail as the underpinning of the correct answer. However, `TreeSet` also implements another interface, `NavigableSet`, that makes the order part of the public contract. Further, the documentation for `TreeSet` states that in a `NavigableSet` implementation, the elements are ordered using their natural ordering or by a comparator provided at set creation time, depending on which constructor is used.

As a result, the order will not vary between implementations and option E is incorrect while option C is correct.

As a final observation, Java's API documentation recommends that whenever possible, natural ordering should be "consistent with equals." The API documentation for `Comparable` states the following: "It is strongly recommended (though not required) that natural orderings be consistent with equals. This is because sorted sets (and sorted maps) without explicit comparators behave 'strangely' when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the `equals` method." Notice that the documentation explicitly calls out the weirdness this question has been investigating.

While this is a reasonable goal for the `Comparable` interface and the “natural ordering,” the whole point of the `Comparator` interface is to permit the expression of multiple



Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers. Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open source or those bundled with the JDK).

Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@halldata.com (phone +1.847.763.9635), who will do whatever they can to help.

Where?

Comments and article proposals should be sent to our editor, **Andrew Binstock**, at javamag_us@oracle.com.

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A, Redwood Shores, CA 94065, USA.

- 👉 Subscription application
- 👉 Download area for code and other items
- 👉 *Java Magazine* in Japanese